This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

Hardware division by small integer constants

H. Fatih Ugurdag, Florent de Dinechin, Y. Serhan Gener,
Sezer Gören, Laurent-Stéphane Didier

Abstract—This article studies the design of custom circuits for division by a small positive constant. Such circuits can be useful to specific FPGA and ASIC applications. The first problem studied is the Euclidean division of an unsigned integer by a constant, computing a quotient and a remainder. Several new solutions are proposed and compared against the state-of-the-art. As the proposed solutions use small look-up tables, they match well with the hardware resources of an FPGA. The article then studies whether the division by the product of two constants is better implemented as two successive dividers or as one atomic divider. It also considers the case when only a quotient or only a remainder are needed. Finally, it addresses the correct rounding of the division of a floating-point number by a small integer constant. All these solutions, and the previous state-of-the-art, are compared in terms of timing, area, and area-timing product. In general, the relevance domains of the various techniques are very different on FPGA and on ASIC.

Index Terms—Integer constant division, IP core generation, parameterized HDL generator, low latency combinational circuit, FPGA synthesis, ASIC synthesis.

1 Introduction

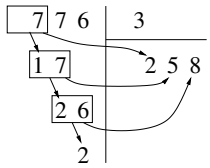
This article considers division by a small integer constant and demonstrates operators for it that can be more efficient than approaches based on standard division [1] or on multiplication by the inverse [2], [3], [4].

1.1 Motivation

Division by a small integer constant is an operation that occurs often enough to justify investigating a specific operator for it. For instance, the core of the Jacobi stencil algorithm computes the average of 3 values: this involves a division by 3. Small integer constants are quite common in such situations. Division by 5 also occurs in decimal-binary conversions. Hardware Euclidean division by a small integer constant can also be used to interleave memory banks in numbers that are not powers of two: if we have D memory banks, an address A must be translated to address A/D in bank $A \mod D$. Finally, a motivation of the present work is the emerging field of High Level Synthesis, which studies the compilation into hardware of applications written in classical sequential languages. There, division by constants will happen in all sort of situations, and it is interesting to provide optimized architectures in such cases.

- *H.F. Ugurdag is with the Dept. of Electrical Eng., Ozyegin University, Istanbul, Turkey.*
  E-mail: fatih.ugurdag@ozyegin.edu.tr
- *F. de Dinechin is with Institut National des Sciences Appliquées, Lyon, France.*
  E-mail:Florent.de-Dinechin@insa-lyon.fr
- *Y.S. Gener and S. Gören are with the Dept. of Computer Eng., Yeditepe University, Istanbul, Turkey.*
  E-mail:{sgener, sgoren}@cse.yeditepe.edu.tr
- *L.-S. Didier is with Université du Sud Toulon Var, France.*
  E-mail: Laurent-Stephane.Didier@univ-tln.fr

Manuscript received October 28, 2016; revised xxxxx.



We first compute the Euclidean division of 7 by 3. This gives the first digit of the quotient, here 2, and the remainder is 1. We now have to divide 176 by 3. In the second iteration, we divide 17 by 3: the second quotient digit is 5, and the remainder is 2. The third iteration divides 26 by 3: the third quotient digit is 8, the remainder is 2, and this is also the remainder of the division of 776 by 3.

Fig. 1. Illustrative example: division by 3 in decimal

1.2 An introductory example

Let us introduce the proposed family of techniques with the help of usual decimal arithmetic. Suppose we want to divide an arbitrary number, say 776, by 3. Fig. 1 describes the paper-and-pencil algorithm in this case.

The key observation is that, in this example, the iteration body consists of the Euclidean division by 3 of a 2-digit decimal number. The first of these two digits is a remainder from previous iteration: its value is 0, 1, or 2, but no larger. We may therefore implement this iteration with a look-up table (LUT), which, for each value from 00 to 29, gives the quotient and remainder of its division by 3. This small LUT will allow us to divide numbers of arbitrary size by 3.

1.3 Related work

Division by a constant in a hardware context has actually been studied quite extensively [2], [3], [4], [5], with good surveys in [4], [6], [7]. There are two main families of techniques: those based on additions/subtractions and those based on multiplication by the reciprocal. The table-based techique studied in this article was introduced in [8]. Prior to that, it had, to our knowledge, only been described in lecture notes [9] as an example of combinational circuit. It is in essence a straightforward adaptation of the paper-and-pencil division algorithm in the case of small divisors. The reason why this technique is not mentioned in the literature

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

is probably that the core of its iteration itself computes a (smaller) division: it does not reduce to either additions or multiplications. However, it is possible to express this smaller division as a small LUT.

Tabulating a complex function is sometimes an efficient technique, and it is the case here. In particular, the proposed technique is very well suited to modern FPGAs, whose reconfigurable logic resources are based on 4- to 6-input LUTs. Nonetheless, this technique is also evaluated in this article for application-specific integrated circuits (ASICs). It is well-known that the optimal architecture for addition or multiplication can be very different on these two targets, ASIC and FPGA. One contribution of this article is to show quantitatively that this is also true for table-based constant division methods.

The proposed architectures will be compared with the state-of-the-art of reciprocal-based architectures of [4], which builds upon [10]. This architecture is called **Recip** in our paper. [4] shows how to determine three integers A, B, and p such that

$$\left\lfloor \frac{X}{D} \right\rfloor = \left\lfloor \frac{AX+B}{2^p} \right\rfloor \quad \forall X \in \{0,\ldots,2^{n-1}\} \tag{1}$$

More specifically, it first determines the minimal bitwidth w of A such that Eq. (1) is possible. There are two possible choices for w in [4], and the smallest one is chosen. In the present work, the value of w chosen is not always the smallest one, but the one that minimizes the area of the corresponding multiplier by A. Otherwise, our Recip reimplementation is faithful to [4], which indeed provides the minimal value of w. We expect Recip to have a timing complexity of roughly $\log(n)$ and an area complexity of n^2.

In software, division by a constant is best performed through multiplication by the inverse. For instance, although Intel processors have always included division instructions, optimizing compilers for these processors (we tried GCC and CLang) will implement the division by 3 of an unsigned 32-bit integer by multiplication by 0xAAAAAAAB followed by a shift. This magic constant is simply $(2^{33}+1)/3$, i.e., the best approximation to $1/3$ that fits on 32 bits. A similar technique is used for signed integers, and even for floating-point numbers [11]. In other words, compilers also use Eq. (1), but minimize p among the few values offered by the architecture (8, 16, 32, and 64 bits on a recent Intel processor). For instance, it turns out that the $32 \times 32 \rightarrow 64$-bit multiplier is adequate for most 32-bit integer divisions.

However, not all processors offer the necessary multiplier. In the Xilinx MicroBlaze 32-bit soft-core processor, the compiler generates a call to a software integer division routine (tens of cycles), unless invoked with the compiler option `-mxl-multiply-high`. This option requires that the processor is built with the optional instruction to recover the high part of a multiplication. Then, the constant division still requires 4 instructions (and a few more cycles).

The hardware dividers reviewed in this article all have much shorter latency than the software solution. Also note that the latency advantage of the hardware solution is greater when both quotient and remainder are needed.

1.4 Outline of the article

Section 2 adapts the radix-10 algorithm demonstrated on Fig. 1 to a radix that is a power of two. On LUT-based FPGAs, this radix is chosen so that the algorithm's LUTs match well with hardware LUTs in FPGAs. The linear architecture obtained by unrolling this recurrence was introduced in [8] and is called **LinArch** throughout this paper.

Section 3 studies variations of this recursion that lead to binary tree implementations with shorter latency. These architectures are called **BTCD** (for Binary Tree Constant Divider) and generalize those introduced in [12].

Section 4 compares the area and delay of all the architectures that compute quotient and remainder.

Section 5 considers the case when only the quotient, or only the remainder, are needed.

Section 6 studies the case when the divider is a product of two small numbers.

Finally, Section 7 studies the division by a small constant of a floating-point input. It shows that it is possible to ensure correct rounding to the nearest with very little overhead.

1.5 Methodology

All methods presented here have been implemented in two hardware generators. One is the open-source project FloPoCo[1]. The object of FloPoCo is application-specific arithmetic for FPGA computing, and this generator has been used to obtain most FPGA results. The other one is a Verilog generator written in Perl at Ozyegin and Yeditepe Universities[2]. Both generators are completely parameterized and can generate circuits for any dividend size and divisor value, and power-of-two radix. Each generator also includes a test framework that has been used to validate the generated architectures.

The FPGA area (A) and delay (T) results in this paper are place and route results obtained with Xilinx Vivado (version 2016-4), targetting a high-speed Kintex-7 (part 7k70tfbv484-3). To compare area and delay of combinatorial architectures, we wrap them between registers and set the target frequency very low, at 1 MHz. This ensures that the tools do not attempt to trade off area for delay for any of the syntheses, thus enabling a fair comparison of latencies.

The A and T results for ASIC have been obtained with Synopsys Design Compiler (version J-2014.09-SP2) using the worst-case version of TSMC ARM 28-nm standard-cell library with a wire-load model. The inputs and outputs are connected to flip-flops in a wrapper module, and the synthesis optimizes for the critical path.

2 Basic recurrence and linear architecture

Let D be the constant divisor, and let k be a small integer. We will use the representation of the input dividend X in radix 2^k, which may also be considered as breaking down the binary decomposition of X into m chunks of k bits (see Fig. 3):

1. http://flopoco.gforge.inria.fr/
2. http://github.com/nemesyslab/ConsDiv/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

TABLE 1
Notations used in this article

X, D, Q, R	dividend, divisor, quotient, and remainder such that $X = DQ + R$ with $0 \leq R < D$
k	size of a chunk of X, i.e., X and Q are considered in radix 2^k
r	size in bits of $D-1$
X_i, Q_i, etc.	sub-words (or: digits in radix 2^k) of X, Q, etc.
n	number of bits of the dividend X
m	number of radix-2^k digits of the dividend X
ℓ	nominal LUT input size in the target FPGAs

$$X = \sum_{i=0}^{m-1} X_i.2^{ki} \quad \text{where} \quad X_i \in \{0, ..., 2^k - 1\} \qquad (2)$$

In this article, we assume that D is not a multiple of 2, as division by 2 reduces to a constant shift, which is simply wiring in a circuit handling binary data.

2.1 Algorithm

The following algorithm computes the quotient Q and remainder R of the high radix Euclidean division of X by constant D. In each step of this algorithm, the partial dividend, Y_i, the partial remainder, R_i, and one radix-2^k digit of the quotient, Q_i, are computed.

Algorithm 1 LUT-based computation of X/D

```
1: procedure ConstantDiv(X, D)
2:     R_m ← 0
3:     for i = m − 1 down to 0 do
4:         Y_i ← X_i + 2^k R_{i+1}
5:                                   (This + is a concatenation)
6:         (Q_i, R_i) ← (⌊Y_i/D⌋, Y_i mod D)
7:                                   (read from a table)
8:     end for
9:     return (Q = Σ_{i=0}^{m−1} Q_i.2^{ki},  R = R_0)
10: end procedure
```

The line $Y_i \leftarrow X_i + 2^k R_{i+1}$ is simply the concatenation of a remainder and a radix-2^k digit. This corresponds to "dropping a digit of the dividend" in Fig. 1.

Let us define r as bitwidth of the largest possible remainder:

$$r = \lceil \log_2(D-1) \rceil \qquad (3)$$

Note that r is also the bitwidth of D, as D is not a power of two. Then, Y_i is of size $k + r$ bits. The second line of the loop body, $(Q_i, R_i) \leftarrow (\lfloor Y_i/D \rfloor,\ Y_i \mod D)$, computes a radix-$2^k$ digit and a remainder: it may be implemented as a LUT with $k + r$ bits of input and $k + r$ bits of output (Fig. 2).

Theorem 1. *Algorithm 1 computes the Euclidean division of X by D: It outputs the quotient Q and the remainder R so that $X = Q \times D + R$. The radix-2^k representation of the quotient Q is also a binary representation, each iteration producing k bits of this quotient.*

Proof. The proof proceeds in two steps. First, Lemma 1 states that $X = D \times \sum_{i=0}^{m} Q_i.2^{-ki} + R_0$. This shows that we compute some kind of Euclidean division, but it is not enough: we also need to show that the Q_i form a binary representation of the result. For this, it is enough to show that they are radix-2^k digits, which is established through Lemma 2.

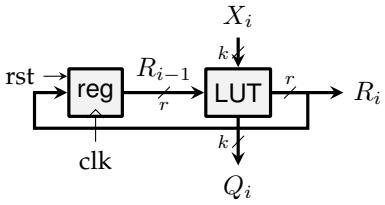


Fig. 2. Sequential architecture for Algorithm 1: LUT-based division of a number written in radix-2^k by a constant

Lemma 1.

$$X = D \sum_{i=0}^{m} Q_i.2^{-ki} + R_0$$

Proof. By definition of Q_i and R_i we have $Y_i = DQ_i + R_i$.

$$\begin{aligned} X &= \textstyle\sum_{i=0}^{m-1} X_i.2^{-ki} \\ &= \textstyle\sum_{i=0}^{m-1} (X_i + 2^k R_{i+1}).2^{-ki} \\ &\quad \textstyle - \sum_{i=0}^{m-1} (2^k R_{i+1}).2^{-ki} \qquad \text{and} \\ &= \textstyle\sum_{i=0}^{m-1} (DQ_i + R_i).2^{-ki} \ - \sum_{i=1}^{m} R_i.2^{-ki} \\ &= D \textstyle\sum_{i=0}^{m-1} Q_i.2^{-ki} \ + R_0 - R_m.2^{-km} \end{aligned}$$

$R_m = 0$. □

Lemma 2. $\forall i \quad 0 \leq Y_i \leq 2^k D - 1$

Proof. The digit X_i verifies by definition $0 \leq X_i \leq 2^k - 1$; R_{i+1} is either 0 (initialization) or the remainder of a division by D, therefore $0 \leq R_i \leq D-1$. Therefore $Y_i = X_i + 2^k R_{i+1}$ verifies $0 \leq Y_i \leq 2^k - 1 + 2^k(D-1)$, or $0 \leq y_i \leq 2^k D - 1$. □

We deduce from the previous lemma and the definition of Q_i as quotient of Y_i by D that

$$\forall i \quad 0 \leq Q_i \leq 2^k - 1$$

which shows that the Q_i are indeed radix-2^k digits. Thanks to Lemma 1, they are the digits of the quotient. □

The algorithm computes k bits of the quotient in each iteration: the larger k is, the fewer iterations are needed for a given input number with bitwidth n. However, the larger k is, the larger the required LUT. Section 2.3 quantifies this trade-off.

2.2 Iterative or unrolled implementation of the basic recurrence

The iteration may be implemented sequentially as depicted in Fig. 2 or as the fully unrolled architecture depicted in Fig. 3. In all of the following, we will focus on the latter, which we denote by LinArch, short for linear architecture, because it enables high-throughput pipelined implementations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

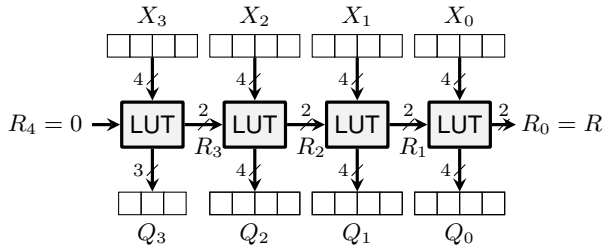


Fig. 3. Unrolled architecture (LinArch) for Algorithm 1: LUT-based division by 3 of a 16-bit number written in radix 2^4 ($k = 4$, $r = 2$)

2.3 Cost evaluation of LinArch

Algorithm 1 performs $\lceil n/k \rceil$ iterations. For a given D and k, the architecture therefore grows linearly with the input size n. Note that general division or multiplication architectures grow quadratically.

Let us now consider how it grows with the constant D.

The area of a LUT in ASIC is essentially proportional to the number of bits it holds. One LUT of Fig. 2 and 3 needs to store $D{\cdot}2^k$ entries of $r{+}k$ bits, or $D{\cdot}2^k(r{+}k)$ bits. To remove D from this formula, we may round up the table size to a power of two: one table will store $2^{r+k}(r+k)$ bits, some of which are "don't care" values. The latter can be optimized out by synthesis tools. Table 2 justifies removing D from the equation: it shows that the tools are able to considerably reduce the area compared to the predicted area, however different D (5 and 7) with the same r and k lead to exactly the same area and delay results.

Finally, the area cost of the sequential architecture grows as $2^{r+k}(r+k)$, and that of LinArch as $\lceil n/k \rceil 2^{r+k}(r+k)$.

The delay of a LUT is essentially proportional to the number of input bits: the delays of both architectures grow as $\lceil n/k \rceil (r+k)$: it is linear in n (this was obvious in Fig. 3.

Section 3 will introduce a parallel architecture that offers smaller delay, sometimes at the expense of larger area.

2.4 FPGA-specific remarks

The basic reconfigurable logic block of current FPGAs is a small LUT with 4 to 6 inputs and one output. We denote it LUTℓ in the following, with $\ell = 4$ to $\ell = 6$. In our case, these LUTℓ can also be used to build the $(r+k)$-input, $(r+k)$-output LUTs we need.

If the unit cost of FPGA logic is the LUTℓ, there is no point in using tables with fewer than ℓ inputs. The value of k should therefore be such that $r + k \geq \ell$. Then the cost of an $(r+k)$-input, $(r+k)$-output LUT is no longer $2^{r+k}(r+k)$ but $2^{r+k-\ell}(r+k)$.

Finally, the area of LinArch in LUTℓ is $\lceil n/k \rceil 2^{\max(r+k-\ell,0)}(r+k)$.

Therefore, the optimal choice of k in terms of area is the smallest k such that $r + k \geq \ell$. As $r = \lceil \log_2(D-1) \rceil$, the method is very area-efficient for small values of D.

However, in each FPGA family, there are restrictions on LUT utilization. In Altera Stratix IV to 10, the Adaptive Logic Module (ALM) can be used as two arbitrary LUT4, but may also implement two LUT5 or two LUT6 under the condition that they share some of their inputs. For instance, a 6-input, 6-output LUT may be built as 3 ALMs.

TABLE 2
Performance on Kintex-7 of LinArch dividers of a 32-bit value by D

D	r	k	estimated A	synthesis results A	synthesis results T
3	2	4	48L	32L	6.0ns
5	3	3	66L	45L	9.3ns
7	3	3	66L	45L	9.3ns
9	4	2	96L	87L	17.9ns
11	4	2	96L	87L	17.9ns
17	5	1	192L	165L	18.5ns

In Xilinx series 5 to 7, the logic slice includes 4 registers and 4 LUT6s, each of which is fractionable as two LUT5 with independent outputs. The sweet spot here is therefore to build 5-input tables, unless we need to register all the outputs, in which case 6-input tables should be preferred.

We may use, for instance, 6-input LUTs to implement division by 3 ($r = 2$) in radix 16 ($k = 4$), as illustrated by Fig. 3. Implementing the core loop costs 6 LUTs (for a 6 bits in, 6 bits out table). The cost for the complete LinArch for n bits is $\lceil n/4 \rceil \times 6$ LUT6s, for instance 36 LUT6s for 24 bits (single precision), or 78 LUTs for 53 bits (double precision).

Table 2 reports some synthesis results – a general comparison with other division techniques will be provided in the following sections. (Note that "L" in Table 2 is short for LUT.

This table illustrates that the method is mostly suited to small constants: for this FPGA ($\ell = 6$), starting with $D = 17$, we have $k = 1$, so the architecture requires as many LUTs as there are bits in the input. Besides, the size of these LUTs then grows as the exponential term 2^{r+k-l}.

3 Parallel division

In this section, we present the family of *Binary Tree Constant Division* (BTCD) circuits. BTCD is to LinArch what fast adders are to carry-propagate adders: its latency is expected to have near logarithmic growth in n (versus the linear growth of LinArch), however, its area is typically larger and is supposed to have growth proportional to $n \log(n)$ (versus again a linear growth for LinArch).

There are 6 fundamental variants of BTCD. On the one hand, a BTCD can be naive, timing-driven, or area-driven (i.e., 3 options). On the other hand, it may be based on regular adder or carry-save adder (i.e., 2 options). When these are paired, we get $3 \times 2 = 6$ options in total. The architecture proposed in [12] is "BTCD naive with regular adders", and we call it simply BTCD. Timing-driven, area-driven, and carry-save are indicated with 't', 'a', and 'c' suffixes, respectively. Hence, the newly proposed versions are BTCDt, BTCDct, BTCDa, BTCDca, and BTCDc.

3.1 The basics of BTCD

Let us first illustrate the method with an example in decimal: Fig. 4 shows the division by 7 of $X = 99767523$.

The dividend X is first subdivided into its m digits. In the example, radix is 10 and $m = 8$:

$$X = X_8 = \sum_{i=0}^{7} X_{8,i} 10^i \tag{4}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

Then each digit $X_{8,i}$ is divided by D in parallel, leading to m quotient digits and m remainder digits:

$$X_{8,i} = DQ_{8,i} + R_{8,i} \tag{5}$$

therefore (4) becomes

$$X_8 = D\sum_{i=0}^{7} Q_{8,i}10^i + \sum_{i=0}^{7} R_{8,i}10^i \tag{6}$$

In Fig. 4, the $Q_{8,i}$ are written on the right-hand side, and can be considered a single decimal number $Q_8 = \sum_{i=0}^{7} Q_{8,i}10^i = 22060523$. The $R_{8,i}$ are written on the left-hand side, and can similarly be considered a single decimal number $R_8 = \sum_{i=0}^{7} R_{8,i}10^i = 11101000$, so (4) can be simply written as:

$$X_8 = D \times Q_8 + R_8 \tag{7}$$

Now, let us define $X_4 = R_8$, and let us group its digits two by two: $X_4 = 22\ 06\ 05\ 23$. This is simply a rewriting in a radix-100 representation:

$$\begin{aligned} X_4 = R_8 &= \sum_{i=0}^{7} R_{8,i}10^i \\ &= \sum_{i=0}^{3} (10R_{8,2i+1} + R_{8,2i})100^i \\ &= \sum_{i=0}^{3} X_{4,i}100^i \end{aligned} \tag{8}$$

Since each radix-10 digit $R_{8,i}$ is a remainder of the division by 7, we have

$$0 \le X_{4,i} \le 66 \tag{9}$$

The next step is to write the Euclidean division of X_4 by D. Again, this can be achieved digit-by-digit:

$$\begin{aligned} X_4 &= \sum_{i=0}^{3} X_{4,i}100^i \\ &= \sum_{i=0}^{3} (DQ_{4,i} + R_{4,i})100^i \\ &= D\sum_{i=0}^{3} Q_{4,i}100^i + \sum_{i=0}^{3} R_{4,i}100^i \\ &= DQ_4 + R_4 \end{aligned} \tag{10}$$

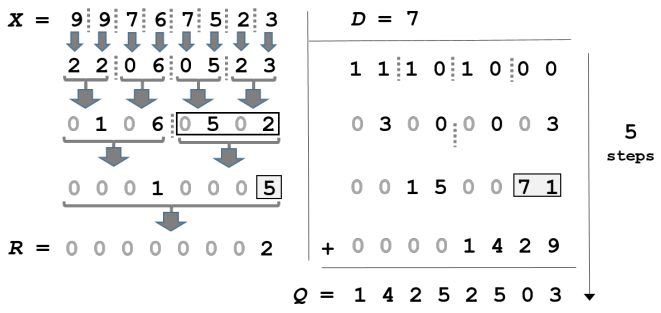


Fig. 4. BTCD's algorithm is shown with radix-10 and $D = 7$

From (9), the Euclidean division by 7 of each $X_{4,i}$ digit will lead to a quotient Q_4 smaller than 10 (hence fitting on one decimal digit), and a remainder between 0 and 6 (also fitting one decimal digit). This explains the apparition of zeroes on both sides of Fig. 4, where $Q_4 = 03000003$ and $R_4 = 01060502$.

Finally, we can again pair two radix-100 digits to obtain a radix-10000 representation of Q_4 and $R_4 = X_2$, and start again: (7) can be rewritten as

$$\begin{aligned} X_8 &= D \times Q_8 + X_4 \\ &= D \times Q_8 + (D \times Q_4 + X_2) \\ &= D \times Q_8 + D \times Q_4 + D \times Q_2 + D \times Q_0 + R_0 \\ &= D \times (Q_8 + Q_4 + Q_2 + Q_0) + R_0 \\ &= D \times Q + R \end{aligned} \tag{11}$$

As $0 \le R_0 < D$, we deduce that Q computed this way is indeed the quotient of the division of X by D.

3.2 Architecture

All this can be generalized it to a dividend of 2^m digits in radix-2^k.

$$\begin{aligned} X = X_{2^m} &= D \times \sum_{j=0}^{m} \sum_{i=0}^{2^j-1} Q_{2^j,i}(2^k)^{i2^{m-j}} + R_0 \\ \text{while} \quad X_{2^{j-1},i} &= R_{2^j,2i+1}(2^k)^{2^{m-j}} + R_{2^j,2i} \end{aligned} \tag{12}$$

The main idea of BTCD is that each Euclidean division in this process can be performed in a LUT: although the radix is squared at each level, the digits manipulated have a constant number of non-zero digits (two decimal digits in our example). Therefore, every arrow in Fig. 4 corresponds to a LUT with the same number of non-zero input bits.

BTCD actually consists of two parallel binary trees. The first is the binary tree of arrows on the left in Fig. 4, where every arrow is a LUT. The first level of LUTs is named iLUT for "initial LUT" (see Fig. 5). The LUTs of other levels are named rLUT for "remainder LUT" (see Fig. 6). Every LUT outputs a quotient (Q) and remainder (R). The R's go to next row, while the Q's are sent to the right, to be added by another binary tree, this time an adder tree, whose structure matches the left tree.

The single binary tree topology shown in Fig. 6 is obtained by overlaying the left and right binary trees described above. In this combined tree, the first level is composed of iLUTs. The nodes of the other levels are called cBLK, short for "combiner BLocK". They are depicted by Fig. 6.

The input and output bitwidths of cBLKs grow as we go down the binary tree from iLUTs. Internally, the number of words in rLUTs is the same in all cBLKs but their output wordsize grows. The adder width grows as well.

In Fig. 5, the subscript of a submodule shows for which bit range of the dividend it does division. Similarly, in Fig. 6, the subscripts of Q and R show for which bit range of the dividend they are respectively the quotient and remainder. As shown in Fig. 5, cBLK15:0 receives a total of 9 bits of input from each of the parent LUTs, namely, cBLK15:8, and cBLK7:0. cBLK7:0 produces Q_R and R_R of Fig. 6, which are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

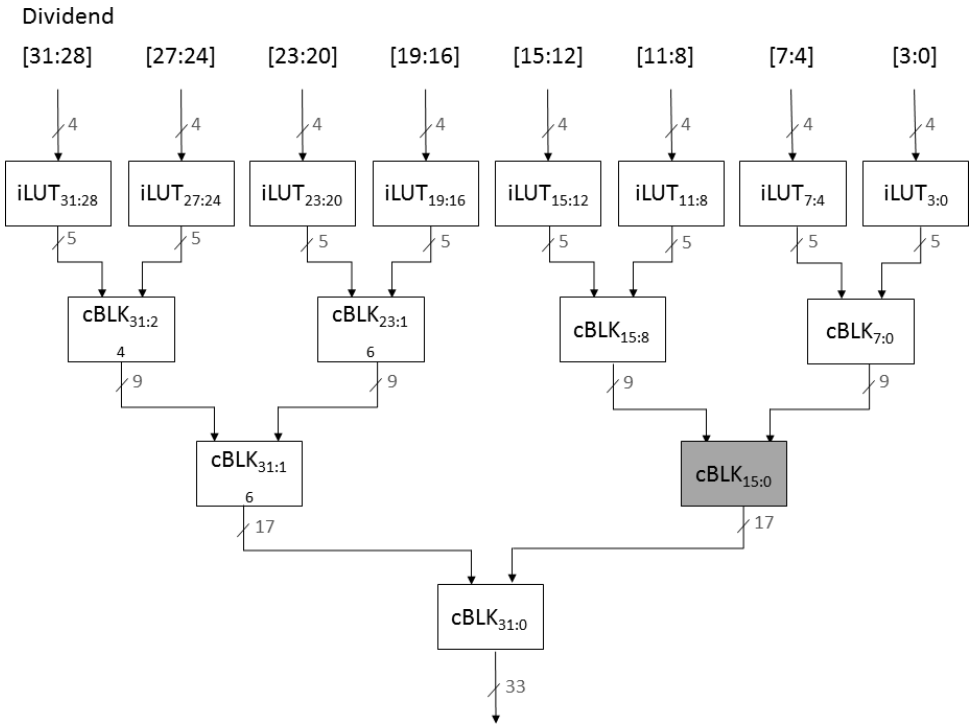


Fig. 5. BTCD circuit for the division by $D = 7$ of a 32-bit binary number initially decomposed in hexadecimal ($k = 4$)

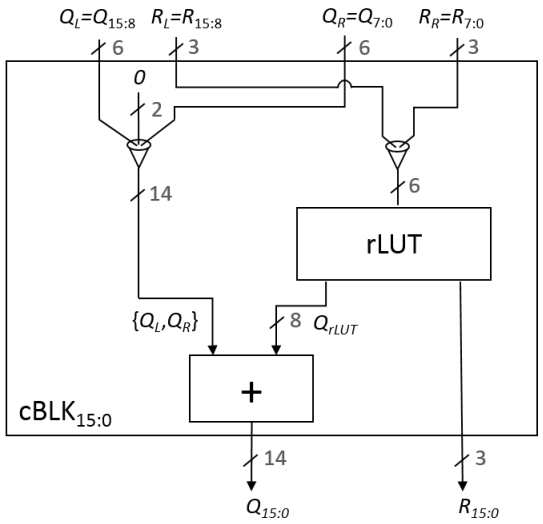


Fig. 6. Internals of $\text{cBLK}_{15:0}$ of Fig. 5 (shaded)

respectively the quotient and remainder that result from the bits [7:0] of the dividend. An 8-bit binary number divided by 7 can at most be 6 bits. On the other hand, remainder out of all subblocks can be at most 3 bits (maximum value of 6). The total number of bits that go from cBLK7:0 to cBLK15:0 is hence 9 bits (6+3).

An iLUT block in Fig. 5 is simply a LUT that inputs k bits and holds 2^k words, each composed of a quotient on $\lceil \log_2 \frac{2^k-1}{D} \rceil$ bits and a remainder on r bits.

An rLUT, on the other hand, has an input of $2r$ bits. The left r bits (R_L) come from the left parent in the binary tree, and the right r bits (R_R) come from the right parent. At level s ($s = 0$ corresponding to the iLUT), this combined input actually represents the number $2^{k2^s} R_L + R_R$, where R_L is shifted left by $k2^s$ bits with respect to R_R: this is a $k2^s + r$-bit number. It outputs the quotient and the remainder of the division of this number by D. The quotient always fits in $k2^s$ bits: its size doubles at each level. However, the number of words stored in an rLUT is D^2 since there are only D possible value for each input remainder: it is constant for each level and remains small for small values of D. Altogether, the number of bits stored in an rLUT is therefore $D^2(k2^s + r)$.

3.3 Naive BTCD

The BTCD generator and the FloPoCo based generator can manage arbitrary size inputs. The initial chunk size k is an input to the generator; its choice will obviously impact the latency/area trade-off.

In the general case, the binary tree is not perfectly balanced. This is easily managed: when there is an odd number of nodes in one level of the binary tree, the leftmost pair (quotient and remainder) is transfered directly to the next level. The interested reader may find the relevant details in the open-source code.

However, the observation that most cases lead to unbalanced binary trees suggests that in such cases, the architecture could be improved by attempting to rebalance the tree. For example, it is possible to reconsider the design choice of splitting the input X into chunks of identical sizes.

Therefore, the implementation described so far will now be called "naive BTCD". The following studies alternative constructions of BTCD trees. It replaces the bottom-up naive BTCD construction with a top-down approach using divide and conquer.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

3.4 BTCD with optimized partitioner

A BTCD partitioner recursively splits the n-bit input X into two chunks of respectively n_{Left} and n_{Right} bits, such that:

$$n = n_{Left} + n_{Right} \tag{13}$$

It then attemps to optimize the BTCD trees on both sides. Two variants of this optimized partitioner are studied: timing-driven and area-driven. The ideas behind both are the same, only the optimization criteria are different.

Division can either be implemented by a simple iLUT or by a BTCD tree. Therefore, the area or the timing of the size-n circuit partitioned can be formulated as $f(n)$:

$$f(n) = \min(f_{iLUT}(n), \min_{n_{Left}}\{g(n_{Left}, n_{Right})\}) \tag{14}$$

Above, $f(n)$ is a recursive function, which recurses indirectly through function g. It computes the minimum-area solution when $f = f_A$, $f_{iLUT} = A_{iLUT}$, and $g = g_A$, while it computes the minimum-timing solution when $f = f_T$, $f_{iLUT} = T_{iLUT}$, and $g = g_T$. Therefore, the only difference between area-driven and timing-driven partitioners is in function g and f_{iLUT}.

3.4.1 Area-driven optimization

The combined divider is composed of the left divider (n_{Left} bits of input), right divider (n_{Right} bits of input), and cBLK (combiner circuit). Hence, the total area equals the sum of the areas of the left divider ($f_A(n_{Left})$) and right divider ($f_A(n_{Right})$) plus the area of cBLK block shown in Fig. 6 ($A_{cBLK}(n_{Left}, n_{Right})$, a non-recursive function).

This leads to the area-specific formulation:

$$\begin{aligned} &g_A(n_{Right}, n_{Left}) = \\ &\quad A_{cBLK}(n_{Left}, n_{Right}) + f_A(n_{Left}) + f_A(n_{Right}) \end{aligned} \tag{15}$$

3.4.2 Timing-driven optimization

The timing version of the formulation is more complicated, because Q and R, at the output of cBLK, have different settling times. Therefore, g_T and f_T are 2D vectors:

$$\begin{aligned} g_T(n_{Right}, n_{Left}) &= \{g_T^R, g_T^Q\} \\ f_T(n) &= \{f_T^R, f_T^Q\} \end{aligned} \tag{16}$$

The $\min$ operations in (14), in the case of timing, must select the min of a number of 2D vectors. One component of each vector is the timing of R, and the other is the timing of Q. Given the design of cBLK (Fig. 6), the timing of Q is always the larger. When the divider is implemented as an iLUT, both timing components are equal. Therefore, this $\min$ can be simply based on the Q timing.

As can be seen in Fig. 6, the timing g_T^R of the R output of cBLK can be computed by adding T_{rLUT} to the $\max$ of $f_T^R(n_{Left})$ and $f_T^R(n_{Right})$:

$$\begin{aligned} g_T^R(n_{Right}, n_{Left}) = &T_{rLUT}(n_{Left}, n_{Right}) \\ &+ \max(f_T^R(n_{Left}), f_T^R(n_{Right})) \end{aligned} \tag{17}$$

This also gives us the timing of Q output of cBLK's rLUT, which enters cBLK's adder. Therefore, the timing of the Q output of cBLK can be computed by finding the input arriving the latest, hence the $\max$ operation in Eq. (18):

$$\begin{aligned} &g_T^Q(n_{Right}, n_{Left}) = \\ &\quad T_{Add}(n_{Left}, n_{Right}) \\ &\quad + \max(g_T^R(n_{Right}, n_{Left}), f_T^Q(n_{Left}), f_T^Q(n_{Right})) \end{aligned} \tag{18}$$

3.4.3 Partitioning dynamic programming algorithm

Now, we will discuss the partitioning algorithm, which is summarized by Eq. (14). A size-n divider can be implemented n different ways at every level of the binary tree. It can be implemented directly by an iLUT (i.e., no partitioning) or n_{Left} can be any of $\{n-1, n-2, .., 2, 1\}$ with corresponding n_{Right}s of $\{1, 2, .., n-2, n-1\}$. Interestingly, the search set of possible partitions can be significantly narrowed down. Consider a partition of $(n_{Left} = a, n_{Right} = b)$ versus $(n_{Left} = b, n_{Right} = a)$, where $a \geqslant b$. The two circuits are identical except for the final cBLK as they both contain a size-a and size-b divider. As for the cBLK, the rLUTs contain the same number of entries (D^2) but their wordsizes are different ($b + r$ bits for (a, b) partition versus $a + r$ bits for (b, a)). Also, the cBLK of (a, b) adds an n-bit number with a b-bit number, while (b, a) adds an n-bit number with an a-bit number. Thus, the cBLK of (a, b) would be definitely smaller than (b, a) and slightly faster (due to the adder). Consequently, it is enough to consider partitions with $a \geqslant b$. This reduces the possibilities by around half at each stage of partitioning. Also, we do not consider iLUTs (i.e., no partitioning) when $n \geqslant 7$ as area starts to blow up although better timing may be obtained.

The problem at hand is not an exponential recursive search. It is a *dynamic programming* problem. Since the left divider depends on only n_{Left}, right divider on n_{Right}, cBLK depends on only n_{Left} and n_{Right}, we can independently optimize each of the three subblocks. For instance, using a non-optimal left divider will not allow us to design a further optimized right divider or a further optimized cBLK. Therefore, when we arrive at a subproblem of $f(a)$ and solve it optimally, we may save it and reuse it when we arrive at the same problem in the recursion tree again.

To report results for up to $n = 128$, we produced the optimal trees (separately for area and timing) starting from $n = 1$ and going up to $n = 128$ with an increment of 1. Larger n use the solutions for the smaller n values. In this approach, for each n, we just compute top-level partition decision, as the lower-level partitioning decisions have already been solved.

It now remains to define the A and T functions used in the previous equations.

3.4.4 Area and delay estimators for ASIC

A LUT of w words of p bits is modelled as p parallel binary trees of MUX2s. The first level of each tree is reduced to wires 75% of the time and to an inverter 25% of the time due to logic synthesis. The LUT area in terms of two-input gates can thus be estimated as $(\frac{49}{16}w - 3)p$.

For the iLUT and rLUT, this yields Eq. (19) and 21:

$$A_{iLUT}(n) = (1 + n)(\frac{49}{16}2^n - 3) \tag{19}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

$$A_{cBLK} = A_{rLUT} + A_{Add} \tag{20}$$

$$A_{rLUT}(n_{Left}, n_{Right}) = (r + n_{Right})(\frac{49}{16}D^2 - 3) \tag{21}$$

$$A_{Add}(n) = ((5/4) \times \lceil \log_2(n) \rceil + 10.5) \times n \tag{22}$$

The timing (in two-input gate delays) is estimated as $4 \times addressBits - \frac{4}{3}$, leading to:

$$T_{iLUT} = 4k - \frac{4}{3} \tag{23}$$

$$T_{rLUT} = 4r - \frac{4}{3} \tag{24}$$

$$T_{Add}(n) = 2 \times \lceil \log_2(n) \rceil + 4 \tag{25}$$

Note that dividing a number smaller than D by D is trivial, and the corresponding LUTs resume to wires (i.e., assumed to be zero area)

For an adder of n-bit numbers, we assume that a prefix-graph based fast adder with Ladner-Fischer topology. We ignore the impact of fanout on area and timing.

The above T and A discussion for the adder in cBLK is for a regular adder. The formulations of A_{Add} and T_{Add} can be easily extended to the version of our design with CSA in cBLK.

3.4.5 Area and delay estimators for FPGAs

On FPGAs, there are two main differences from ASIC:

- A table of w words of p bits costs p LUTℓ as long as $\log_2 w \leq \ell$ (see Section 2.4). If $\log_2 w > \ell$, the cost grows as $p \times w/2^\ell$.
- For the sizes targeted here, the optimal adder architecture in an FPGA is a plain carry-propagation adder exploiting the dedicated fast-carry lines. It has linear delay and area.

In the presented implementation, the required A and T functions are implemented in the Target class hierarchy of the FloPoCo tool. These models are quite accurate in terms of logic delay, although routing delays remain difficult to predict [13]. The details are FPGA-specific and out of scope of this paper. The interested reader will find them in the source code of the tool.

3.5 BTCD with carry-save

Quite often, the critical path of BTCD goes through the adders (chained together). Although all quotients to be summed could be input to a traditional compression tree, we do a 3:2 compression at every level as quotients "arrive" one by one. Chain of regular adders are replaced by 3:2 Carry-Save Adders (CSAs), hence the new cBLK design in Fig. 7. It is still a linear chain but each stage is a CSA with one logic level of parallel full-adder cells instead of a $\log(n)$ complexity adder. There is still a regular final adder. CSAs are placed into cBLKs with the exception of the first level of cBLKs (after the iLUTs). Second level of cBLKs inherit two sums from above and produce a new quotient to make it 3 quotients. Then, every level cBLKs reduces 3 quotients to 2 quotients.

Replacing cBLK's adder with a CSA can be applied to all three kinds of BTCDs, namely, BTCD naive (BTCD becomes BTCDc), BTCD area-driven partitioner (BTCDa becomes BTCDca), and BTCD timing-driven partitioner (BTCDt becomes BTCDct).

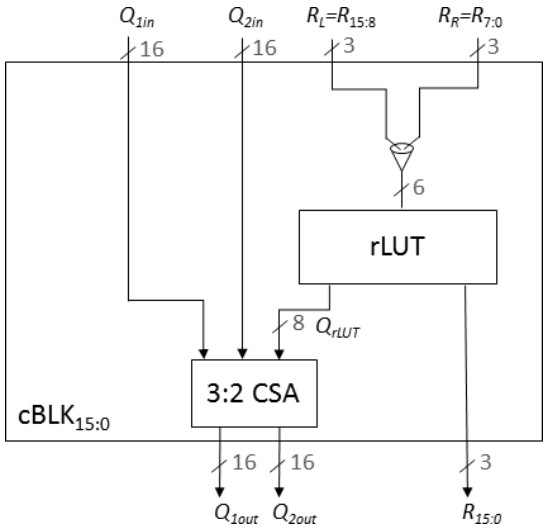


Fig. 7. Block cBLK with the 2-input Adder replaced by a 3:2 Carry-Save Adder

Thanks to the availability of fast-carry logic on virtually all FPGAs, carry-save BTCD is irrelevant for them.

4 Comparison of BTCD with LinArch and Recip method

In order to identify the relevance domains of LinArch, BTCD, and Recip method, this section shows a generous amount of synthesis results of these three families of architectures, first for an ASIC standard-cell library as a target, then for Xilinx Kintex-7 FPGA.

4.1 ASIC synthesis results

For BTCD*, the HDL generation process is a two-phase approach. A tree description is computed by the naive method or a partitioning method, then input to the HDL generator. The tree can even be described manually.

We have generated 6 different BTCDs (hence the BTCD* in Table 3), LinArch, and Recip designs for divisors (D) of 3, 5, 11, and 23. For each divisor, we have generated different versions of the design with a dividend bitwidth (n) of 8, 16, 32, 64, and 128. We also tried 3 chunk sizes (k = 3, 4, 5) for BTCD, BTCDc and 6 chunk sizes for LinArch (k = 1 to 6). The other variants (BTCDt, BTCDtc, BTCDa, BTCDac, and Recip) do not have k as a parameter.

Our synthesis script does a binary search for the smallest latency in 4 synthesis runs for a given circuit. That is, we set a latency constraint and see if it is met; if not, we raise it, if yes, we lower it.

Table 3 summarizes the delay, area, and area-timing product (ATP) results obtained in a total of 1,360 synthesis runs. Area results are critical in cases where pipelining and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

TABLE 3
Timing (T), Area (A), Area-Timing Product (ATP) of BTCD*, Recip, LinArch on 28-nm ASIC

	Best T								Best A						Best ATP				
	LinArch			BTCD*			Recip		LinArch			BTCD*			LinArch		BTCD*		Recip
D = 3	k	T	A		T	A	T	A	k	T	A		T	A	k	ATP		ATP	ATP
n = 8	5	**0.21**	370	ct	0.28	313	0.29	284	1	0.335	**141**	ca	0.308	291	2	**37**	ct	88	81
16	5	**0.31**	550	5	0.41	849	0.44	1074	1	0.676	**322**	ca	0.45	642	3	**122**	ca	289	473
32	6	**0.47**	1006	ct	0.53	1693	0.58	2648	2	0.925	**658**	ca	0.55	1646	5	**385**	c5	879	1536
64	6	0.72	1674	ct	**0.63**	4530	0.72	8199	3	1.244	**1228**	ca	0.675	3487	5	**1084**	ca	2354	5903
128	6	1.31	2573	ct	**0.74**	8783	0.85	30141	5	1.474	**2342**	ca	0.8	7278	6	**3376**	ca	5822	25469
D = 5		$n^{0.7}$			$(\log n)^{1.1}$		$(\log n)^{1.3}$	$n^{1.7}$			n			$n^{0.9}\log n$					
n = 8	4	**0.24**	383	c3	0.27	337	0.32	401	1	0.366	**214**	3	0.27	323	2	**64**	3	87	128
16	5	**0.38**	1091	4	0.43	1060	0.45	1018	1	0.81	**508**	c3	0.504	878	2	**368**	c5	442	462
32	5	0.71	2228	4	0.59	3314	**0.58**	2632	1	1.69	**1112**	ca	0.75	2015	4	1480	c5	**1373**	1527
64	5	1.22	3879	c4	0.73	5444	**0.71**	9011	1	3.48	**2402**	ca	1.006	3593	5	4732	ca	**3615**	6434
128	5	2.45	7279	c4	0.88	12002	**0.85**	30723	1	7.012	**5117**	ca	1.25	8412	5	17819	c5	**9394**	25961
D = 11		$n^{0.8}$			$(\log n)^{1.4}$		$(\log n)^{1.1}$	$n^{1.6}$			$n^{1.1}$			$n^{0.9}\log n$					
n = 8	6	**0.24**	597	c3	0.26	500	0.31	**337**	1	0.402	454	ct	0.264	435	6	143	ct	115	**103**
16	6	**0.45**	2576	4	0.51	2145	0.46	**1157**	1	0.995	1262	a	0.754	1264	2	992	t	839	**526**
32	6	0.83	4557	4	0.71	5380	**0.56**	**2706**	1	2.2	2870	ca	1.6	2942	6	3791	ct	3140	**1521**
64	6	1.59	9493	c4	0.93	11089	**0.68**	8156	1	4.57	**6051**	ca	2.6	6461	6	15046	ct	8976	**5530**
128	6	3.22	16912	c4	1.12	22080	**0.82**	25788	1	9.335	**12126**	ca		12146	6	54389	ct	22337	**21043**
D = 23		$n^{0.9}$			$(\log n)^{1.7}$		$(\log n)^{1.2}$	$n^{1.6}$			$n^{1.2}$			$n^{0.9}\log n$					
n = 8	6	**0.26**	1159	c3	**0.26**	454	0.37	**342**	1	0.366	454	c3	0.262	454	2	129	c3	**119**	127
16	6	0.55	5839	ct	0.53	3395	**0.44**	**1047**	1	1.02	1742	ca	0.9775	1624	2	1354	a	1572	**463**
32	6	1.19	14358	t	0.77	9260	**0.55**	**2261**	1	2.345	4125	ca	1.96	4098	2	7096	c3	6538	**1238**
64	6	2.46	30620	t	1.06	18890	**0.67**	**7049**	1	4.975	8956	ca	3.728	9593	2	32895	c3	18250	**4723**
128	6	5	61031	t	1.33	39678	**0.81**	23448	1	10.272	**17865**	ca	7.265	19772	2	135979	ct	48272	**18946**
		$n^{1.1}$			$(\log n)^{1.9}$		$(\log n)^{0.9}$	$n^{1.5}$			$n^{1.3}$			$n^{1.1}\log n$					

TABLE 4
Comparison of LinArch, BTCD, and Recip Method on Kintex-7 FPGA (L: LUT, BR: BlockRAM)

		LinArch		BTCD		Recip	
D	n	T	A	T	A	T	A
3	8	3.7ns	**8**L	**3.6**ns	12L	3.6ns	17L
	16	**3.6**ns	**17**L	3.7ns	37L	4.5ns	52L
	32	6.0ns	**32**L	**4.8**ns	95L	6.1ns	139L
	64	13.5ns	**63**L	**6.2**ns	225L	8.5ns	346L
	128	26.3ns	**128**L	**8.4**ns	517L	12.4ns	825L
5	8	3.6ns	**9**L	3.6ns	18L	**3.6**ns	20L
	16	4.4ns	**21**L	**3.8**ns	44L	4.9ns	54L
	32	9.3ns	**45**L	**4.7**ns	109L	7.0ns	140L
	64	20.1ns	**93**L	**6.7**ns	270L	8.6ns	346L
	128	38.3ns	**189**L	**9.0**ns	612L	12.0ns	824L
11	8	3.7ns	**15**L	**3.6**ns	20L	4.7ns	29L
	16	8.0ns	**39**L	**3.8**ns	79L	4.8ns	104L
	32	17.9ns	**87**L	**6.1**ns	212L	6.6ns	219L
	64	39.0ns	**183**L	8.8ns	526L	**8.4**ns	503L
	128	76.3ns	**375**L	**8.8**ns	1.5BR + 1100L	12.2ns	1082L
23	8	3.8ns	**21**L	**3.7**ns	36L	4.0ns	26L
	16	7.4ns	**69**L	5.6ns	197L	**5.1**ns	83L
	32	18.5ns	**165**L	**6.8**ns	1BR + 436L	7.3ns	169L
	64	36.6ns	**357**L	**6.5**ns	2.5BR + 959L	9.0ns	401L
	128	72.5ns	**741**L	**6.6**ns	6BR + 2028L	13.4ns	931L

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

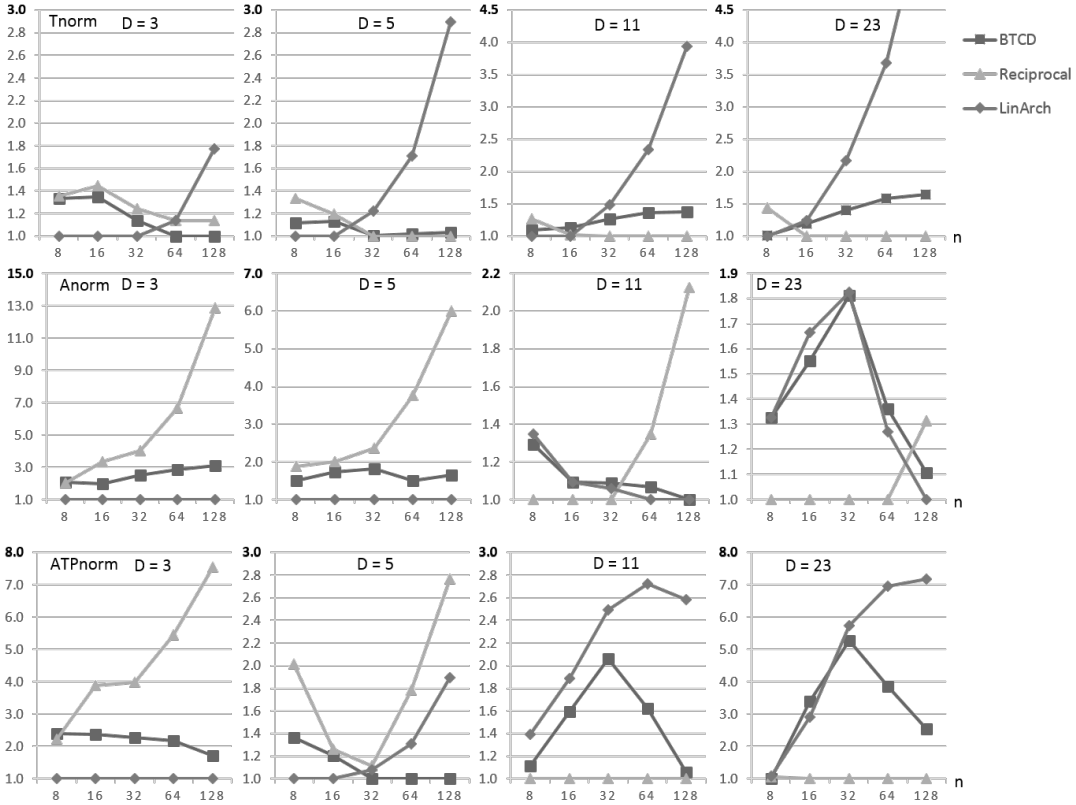


Fig. 8. Plot versions of the results presented in Table 3.

hence multiple-cycle latency is allowed. ATP results have significance as they correlate with energy consumption (because power correlates with area). However, they can also be considered as a more universal area metric for the cases when the circuit is synthesized under more relaxed timing constraints. ATP is also an indicator of power consumption per unit performance.

Table 3 is actually three tables in one. The LinArch, BTCD*, and Recip column on the left form a "Timing (T) table". The Recip, second BTCD*, and second LinArch column in the middle form a "Area (A) table". The third LinArch, third BTCD*, and second Recip column on the right are part of "ATP table".

In Table 3, there are up to 3 different BTCD* designs reported in one row out of the total of 10 possible BTCD* designs, where * can be one of {t, a, 3, 4, 5, ct, ca, c3, c4, c5}[3] (reported on the left of each of the 3 BTCD sections of the table). The BTCD* with the Best A may be a different permutation than the one that yields the Best T or the Best ATP. We report both T and A for the BTCD* with Best T, same applies to the BTCD* with Best A. That is because even when T is the optimization criterion, prohibitive A is undesired. While A is optimized, too large T values should be noted. On the other hand, we only report ATP in the Best ATP column as ATP is already a way of reporting both A and T (though in a single number). LinArch has 6 different versions (with k of 1 to 6), and everything discussed for BTCD* above also applies to LinArch. As for Recip, there is only one version, and its T and A are reported in the middle of the table, while its ATP is reported on the very right.

In Table 3, the best (smallest) value in every category is marked in bold. Table 3 also displays the complexity trend of every type of result for every divisor value in terms of n. These are experimental formulas, and they more or less agree with the theory, which estimates that the T of LinArch, BTCD*, Recip grow proportional to n, $\log(n)$, $\log(n)$, respectively, while their A in proportion to n, $n\log(n)$, and n^2, respectively.

The results in Table 3 are shown as plots in Fig. 8. The vertical axes show "normalized" T, A, ATP values such that the smallest value is taken as 1.0. The trend is that LinArch's T and Recip's A blow up for large n. For small n and small D, LinArch wins in all metrics (A, T, ATP). For small n and large D, Recip usually wins in all metrics. BTCD* usually offers a compromise between T and A.

4.2 FPGA synthesis results

Table 4 shows some synthesis results on Xilinx Kintex-7 obtained thanks to the FloPoCo implementation of the LinARch, naive BTCD and reciprocal methods. In the reciprocal method, instead of choosing the smallest constant size as in [4], this implementation chooses the constant that leads to the smallest architecture (smallest number of *full-adders*). The constant multipliers of FloPoCo are used [14]. They are behind the state-of-the-art [15], but good enough for a fair comparison of the trends.

For large n and large D, the Vivado tool begins using BlockRAM resources (reported as BR) for BTCD, because of

3. t: timing-driven, a: area-driven, 3-5: naive, c: carry-save adder

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

TABLE 6
The overhead of computing the remainder in Recip method. Recip/R denotes the architecture of [4] that computes quotient only.

		Recip/R		Recip	
D	n	T	A	T	A
3	8	3.6ns	17L	3.5ns	16L
	16	4.5ns	52L	4.2ns	51L
	32	6.1ns	139L	6.0ns	138L
	64	8.5ns	346L	7.9ns	345L
	128	12.4ns	825L	13.3ns	824L
23	8	4.0ns	26L	3.7ns	20L
	16	5.1ns	83L	3.8ns	71L
	32	7.3ns	169L	5.8ns	158L
	64	9.0ns	401L	7.9ns	390L
	128	13.4ns	931L	14.0ns	920L

the large output width of the tables. This yields an apple-to-orange comparison of the area.

Here, LinArch is consistently the best option in terms of area, and BTCD offers a better delay for larger n. Recip is a balanced alternative for large n and large d.

5 Remainder-only or quotient-only

This section discusses some variants of the previous architectures that are more efficient by only outputting the remainder or only outputting the quotient.

5.1 Reciprocal method outputting only the quotient

The reciprocal method as published in [4] only computes the quotient. The previous tables report results for architectures that are slightly more complex, as they also compute the remainder as $R = X - DQ$. However, the overhead of this computation is very small, as Table 6 shows. Indeed, this computation has to be only performed on the number of bits of R, which is small with respect to the number of bits of X.

5.2 Remainder-only variant of LinArch and BTCD

In LinArch, if one only needs the remainder, the quotient bits need not be stored at all. This entails savings in terms of area that are easy to predict (roughly a factor $r/(r+k)$, as illustrated by Table 5). However, there is almost no improvement in delay, as the critical path is unchanged (see Fig. 3).

In BTCD, computing only the remainder improves both area and delay: if the quotient is not needed, there is no need for the corresponding part of the table nor the addition tree. All that remains is a binary tree of tables that have $2r$ inputs and r outputs. Note that there is a binary tree in [7] for this case, but it still involves additions.

Besides, as the critical path was in the quotient addition tree, the delay of a remainder-only architecture is significantly smaller than the delay of the complete architecture.

As Table 5 shows, BTCD therefore offers a much better area-time trade-off than LinArch if only the remainder is needed. For instance, for $k = 5$, the area is nearly identical (both architectures build the same number of LUTs with 6 inputs and 3 outputs) while the delay of BTCD is logarithmic instead of linear for LinArch.

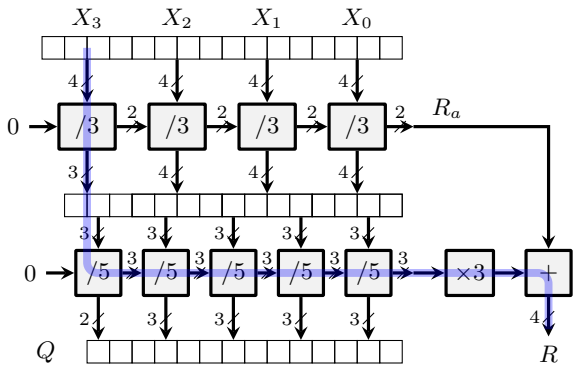


Fig. 9. Composite divider of a 16-bit value by $15 = 3 \times 5$, with the critical path highlighted

6 Composite division

The table-based methods scale poorly with the size (in bits) of the constant. For a large constant that is the product of two smaller ones, it is however possible to divide successively by the two smaller constants. This will often lead to a smaller architecture than a monolithic division by the large constant. The following first formalizes this intuition, then discusses the choice of an optimal factorization.

6.1 Algorithm

Let us first assume that D is the product of two smaller constants:

$$D = D_a \times D_b \tag{26}$$

The Euclidean division of X by D_a can be written

$$X = D_a Q_a + R_a \quad \text{with } R_a \leq D_a - 1. \tag{27}$$

Then we can divide Q_a by D_b:

$$Q_a = D_b Q + R_b \quad \text{with } R_b \leq D_b - 1. \tag{28}$$

Putting all together, we obtain

$$\begin{array}{rcl} X & = & D_a D_b Q + D_a R_b + R_a \\ & = & D \times Q + R \quad \text{where } R = D_a R_b + R_a. \end{array} \tag{29}$$

Since $R = D_a R_b + R_a \leq D_a(D_b - 1) + D_a - 1 = D - 1$, R is indeed the remainder of the division of X by D, and Q_b is indeed the proper quotient.

Now, if the divisor D is the product of more than two factors (i.e. if D_a or D_b can themselves be factored), the previous decomposition can be applied recursively.

6.2 Architecture

In terms of architecture, we need two of the previous dividers to compute the Euclidean divisions by D_a and D_b, plus a multiply-and-add to compute $r = D_a R_b + R_a$.

This additional multiply-and-add is typically small for two reasons. First, the remainders are small. Secondly, this is a constant multiplication, for which a range of techniques exist, some table-based [16], [17], some based on shift-and-add [14], [15], [18].

As can be seen in Fig. 9, the critical path is that of the divider with the largest m, plus one LUT for the quotient output, and a small constant multiplication and addition for the remainder output.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

TABLE 5
Comparison of remainder-only architectures (R) to Euclidean divider architectures (Q+R)

		LinArch				BTCD			
D	n	Q+R		R		Q+R		R	
		T	A	T	A	T	A	T	A
3	8	3.7ns	8L	3.6ns	3L	3.6ns	12L	3.6ns	2L
	16	3.8ns	15L	3.5ns	5L	3.7ns	37L	3.7ns	5L
	32	6.0ns	32L	6.5ns	11L	4.8ns	95L	3.8ns	12L
	64	13.5ns	63L	14.2ns	21L	6.2ns	225L	4.8ns	25L
	128	26.3ns	128L	27.2ns	43L	8.4ns	517L	5.3ns	50L
5	8	3.6ns	9L	3.6ns	6L	3.6ns	18L	3.5ns	7L
	16	4.4ns	21L	4.4ns	14L	3.8ns	44L	3.6ns	14L
	32	9.3ns	45L	9.8ns	30L	4.7ns	109L	3.7ns	29L
	64	20.1ns	93L	21.2ns	62L	6.7ns	270L	4.5ns	62L
	128	38.3ns	189L	37.7ns	126L	9.0ns	612L	5.2ns	129L

6.3 Results and discussions

Table 7 provides synthesis results on FPGA for three constants that are of practical significance: 9 appears in some 2D stencils, 15 is (up to a power of two) 60 and occurs in timing conversions, and 45 is (up to a power of two) 360 which appears in degree/radian conversions. Interestingly, the two latter constants were probably chosen in ancient times because they can be divided by 2, 3, 4, and 5.

Table 7 shows that composite dividers are not only smaller, they are also consistently faster than the atomic LinArch on FPGAs. This is easily explained on the example of division by 9. The atomic LinArch divider by 9 is predicted to require about the same area as the composite divider. However, its optimal value of k is 2, and the 32-bit input X is therefore decomposed in $m = 16$ radix-2^2 digits: the architecture has 16 LUTs on the critical path. When we compose two dividers by 3, optimal value of k is 4 for each, so the architecture of each sub-divider has only $m = 8$ LUTs on the critical path. As Fig. 9 shows, the critical path of the composite architecture is therefore $8+1$ LUTs only (plus the remainder reconstruction) instead of 16 LUTs.

Table 8 shows how composite dividers may improve both area and delay in the case of a 28-nm ASIC standard-cell library.

7 Floating-point division by a small integer constant

When accelerating floating-point computations on FPGAs, it makes sense to use operators that are tailored to the context. This section shows how to build a divider of a floating-point input X by a small integer constant that ensures bit-for-bit compatibility with a IEEE 754-compliant divider. A compiler of floating-point code to FPGA can use this operator as a drop-in replacement for a classical divider, at a fraction of the cost.

The proposed floating-point divider requires a Euclidean divider and can use for this any of the variants of the architectures previously presented (linear, composite, and/or parallel). For simplicity, the results are presented with LinArch.

A floating-point input X is given by its mantissa M and exponent E:

$$X = 2^E X_f \quad \text{with } X_f \in [1, 2) \quad . \tag{30}$$

Similarly, the floating-point representation of our integer divisor D is:

$$D = 2^S D_f \quad \text{with } D_f \in [1, 2) \quad . \tag{31}$$

One may remark that $S = r - 1$ if D is not a power of two.

The main issues to address are the normalization and rounding of the floating-point division of X by D according to the IEEE 754 standard [19].

7.1 Normalization

Let us write the division

$$\frac{X}{D} = \frac{X_f.2^E}{D} = \frac{2^S X_f}{D} 2^{E-S} = \frac{X_f}{D_f} 2^{E-S}. \tag{32}$$

As $\frac{X_f}{D_f} \in [0.5, 2)$, this is almost the normalized mantissa of the floating-point representation of the result:

- if $X_f \geq D_f$, then $\frac{X_f}{D_f} \in [1, 2)$, the mantissa is correctly normalized and the floating-point number to be returned is
$$Y = \circ\left(\frac{2^S X_f}{D}\right) 2^{E-S} \tag{33}$$
where $\circ(z)$ denotes the IEEE-standard rounding to nearest even of a real z.
- if $X_f < D_f$, then $\frac{X_f}{D_f} \in [0.5, 1)$, the mantissa has to be shifted left by one. Thus, the floating-point number to be returned is
$$Y = \circ\left(\frac{2^{S+1} X_f}{D}\right) 2^{E-S-1} \quad . \tag{34}$$

The comparison between X_f and D_f is extremely cheap as long as D is a small integer, because in this case D_f has only r non-zero bits. Thus, the comparison is reduced to the comparison of these r bits to the leading r bits of X_f. As both X_f and D_f have a leading one, we need a comparator on $r - 1$ bits. On FPGAs, this is a very small delay using fast-carry propagation.

7.2 Rounding

Let us now address the issue of correctly rounding the mantissa fraction. If we ignore the remainder, the obtained result is the rounding towards zero of the floating-point division.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2017.2707488, IEEE Transactions on Computers

To obtain correct rounding to the nearest, a first idea is to consider the final remainder. If it is larger than $D/2$, we should round up, *i.e.* increment the mantissa. The comparison to $D/2$ would cost nothing (actually the last table would hold the result of this comparison instead of the remainder value), but this would mean an addition of the full mantissa size, which would consume some logic and have a latency comparable to the division itself, due to carry propagation.

A better idea is to use the identity $\circ(z) = \lfloor z + \frac{1}{2} \rfloor$, which in our case becomes

$$\circ\left(\frac{2^{S+\epsilon}X_f}{D}\right) = \left\lfloor \frac{2^{S+\epsilon}X_f + D/2}{D} \right\rfloor \tag{35}$$

with ϵ being 0 if $X_f \geq D_f$, and 1 otherwise. In the floating-point context we may assume that D is odd, since powers of two are managed as exponents. Let us write $D = 2H + 1$. We obtain

$$\begin{aligned} \circ\left(\frac{2^{S+\epsilon}X_f}{D}\right) &= \left\lfloor \frac{2^{S+\epsilon}X_f + H}{D} + \frac{1}{2D} \right\rfloor \\ &= \left\lfloor \frac{2^{S+\epsilon}X_f + H}{D} \right\rfloor \end{aligned} \tag{36}$$

so instead of adding a round bit to the result, we may add H to the dividend before its input into the integer divisor. It seems we haven't won much, but this pre-addition is actually for free: the addend $H = \frac{D-1}{2}$ is an S-bit number, and we have to add it to the mantissa of X that is shifted left by $S+\epsilon$ bits, so it is a mere concatenation. Thus, we save the adder area and the carry propagation latency.

To sum up, the management of a floating-point input adds to the area and latency of the mantissa divider those of one (small) exponent adder, and of one (large) mantissa multiplexer, as illustrated by Figure 10. On this figure, ξ is a 2-bit exception vector used to represent 0, $\pm\infty$ and NaN (Not a Number).

The implementation in FloPoCo manages such divisions by small integer constants and all their powers of two. The only additional issues are in the overflow/underflow logic (the Exn box on Figure 10), but they are too straightforward to be detailed here.

TABLE 7
Examples of composite 32-bit dividers on Kintex7. Estimates ignore the cost of the remainder reconstruction, which is accounted for in the measured area and delay. In all cases the optimal value of k is used.

D	decomposition	predict. A	meas. A	meas. T
9	atomic Recip	–	184L	6.2ns
	atomic BTCD	–	218L	**6.1ns**
	atomic LinArch	96	87L	17.9ns
	3×3	48+48=96	**65L**	7.0ns
15	atomic Recip	–	97L	6.2ns
	atomic BTCD	–	199L	**5.8ns**
	atomic LinArch	96	87L	17.9ns
	5×3	66+48=114	80L	9.6ns
	3×5	68+46=114	**75L**	9.4ns
45	atomic Recip	–	207L	**8.0ns**
	atomic LinArch	448	371L	24.3ns
	9×5	96+60=156	134L	18.4ns
	5×9	66+90=156	134L	18.3ns
	$5 \times 3 \times 3$	66+48+48=162	**108L**	10.0ns
	$3 \times 3 \times 5$	48+48+60=156	113L	9.8ns
	15×3	96+48=144	120L	18.5ns
	3×15	48+96=144	125L	17.6ns

TABLE 8
Examples of composite 32-bit dividers by 9 on 28-nm ASIC

D	decomp.	A	T
9	atomic (LinArch, k=1)	1966	1.87 ns
	atomic (BTCDca)	2528	1.42 ns
	3×3 (LinArch, k=1)	**1400**	1.50 ns
	3×3 (BTCDc3)	3243	**1.06** ns

TABLE 9
Floating-point division by a constant on Kintex-7

		single precision		double precision	
D	method	T	A	T	A
3	LinArch (mantissa only)	**5.2ns** (4.8ns)	**38L** (26L)	12.0ns (11.8ns)	**88L** (54L)
	BTCD	6.0ns	87L	**7.7ns**	223L
	Recip	6.1ns	142L	8.9ns	364L
	[3]	7.8ns	127L	10.5ns	325L
5	LinArch (mantissa only)	8.0ns (7.3ns)	**56L** (38L)	16.6ns (16.5ns)	**116L** (81L)
	BTCD	**5.8ns**	110L	**8.0ns**	264L
	Recip	6.6ns	142L	8.3ns	358L
	[3]	7.3ns	125L	10.7ns	322L

7.3 Results

Here, we only report FPGA results, because FPGA acceleration of floating-point code is probably the only context where floating-point constant dividers are useful. In Table 9, the lines (mantissa only) illustrate the overhead of floating-point management. As expected, the area overhead is large but the timing overhead is very small, and both are almost independent of D.

This table also compares with the state-of-the-art, and there is another work to mention here: [3] is essentially a method that multiplies by the reciprocal using a shift-and-add tree that exploits the periodic binary representation of $1/D$. It computes the required number of periods to ensure correct rounding of the multiplication by the exact rational $1/D$, and is therefore bit-for-bit compatible with correctly rounded division by D. However, the architectures proposed in the present article offer better performance.

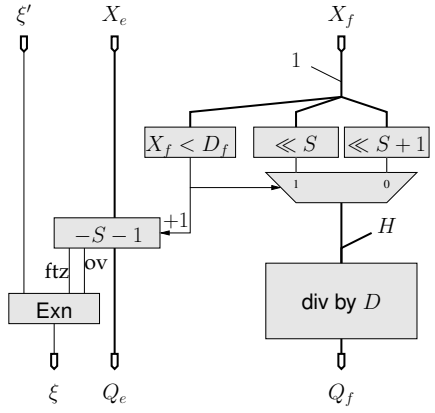


Fig. 10. Floating-point division by a small constant.

8 Conclusion

This article adds division by a small integer constant (such as 3 or 10) to the bestiary of arithmetic operators available to C-to-hardware compilers. This operation can be implemented very efficiently, be it for integer inputs or for floating-point inputs.

The article studies qualitatively the performance (area and delay) of three families of techniques (linear, binary tree and multiplicative). On ASIC, each method has its relevance domain. On FPGAs, the simplest table-based methods behave comparatively better, thanks to the LUT-based hardware structure of FPGAs. Due to increasing routing pressure as technology progresses, the number of inputs to these FPGA LUTs keeps increasing. This should make this technique increasingly relevant in the future.

References

[1] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[2] E. Artzy, J. A. Hinds, and H. J. Saal, "A fast division technique for constant divisors," *Communications of the ACM*, vol. 19, pp. 98–101, Feb. 1976.

[3] F. de Dinechin, "Multiplication by rational constants," *IEEE Transactions on Circuits and Systems, II*, vol. 52, no. 2, pp. 98–102, Feb. 2012.

[4] T. Drane, W. C. Cheung, and G. Constantinides, "Correctly rounded constant integer division via multiply-add," in *Int. Symp. Circuits and Systems (ISCAS)*, May 2012, pp. 1243–1246.

[5] S.-Y. R. Li, "Fast constant division routines," *IEEE Transactions on Computers*, vol. C-34, no. 9, pp. 866–869, Sep. 1985.

[6] P. Srinivasan and F. E. Petry, "Constant-division algorithms," *IEE Proc. Computers and Digital Techniques*, vol. 141, no. 6, pp. 334–340, Nov. 1994.

[7] R. W. Doran, "Special cases of division," *Journal of Universal Computer Science*, vol. 1, no. 3, pp. 67–82, 1995.

[8] F. de Dinechin and L.-S. Didier, "Table-based division by small integer constants," in *Int. Symp. on Applied Reconfigurable Computing (ARC)*, Mar. 2012, pp. 53–63.

[9] A. Paplinski, *CSE2306/1308 Digital Logic Lecture Note, Lecture 8*, Clayton School of Information Technology Monash University, Australia, 2006.

[10] A. D. Robison, "N-bit unsigned division via n-bit multiply-add," in *Int. Symp. on Computer Arithmetic (ARITH)*, 2005, pp. 131–139.

[11] N. Brisebarre, J.-M. Muller, and S. K. Raina, "Accelerating correctly rounded floating-point division when the divisor is known in advance," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 1069–1072, 2004.

[12] H. F. Ugurdag, A. Bayram, V. E. Levent, and S. Gören, "Efficient combinational circuits for division by small integer constants," in *Int. Symp. on Computer Arithmetic (ARITH)*, Jul. 2016, pp. 1–7.

[13] M. Istoan and F. de Dinechin, "Automating the pipeline of arithmetic datapaths," in *Design Automation and Test in Europe (DATE)*, Mar. 2017.

[14] N. Brisebarre, F. de Dinechin, and J.-M. Muller, "Integer and floating-point constant multipliers for FPGAs," in *Int. Conf. on Application-specific Systems, Architectures and Processors*, 2008, pp. 239–244.

[15] M. Kumm, "Multiple constant multiplication optimizations for field programmable gate arrays," Ph.D. dissertation, Kassel Univ., 2016.

[16] K. D. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, no. 10, p. 80, May 1993.

[17] M. J. Wirthlin, "Constant coefficient multiplication using look-up tables," *Journal of VLSI Signal Processing*, vol. 36, no. 1, pp. 7–15, 2004.

[18] O. Gustafsson, "Lower bounds for constant multiplication problems," *IEEE Transactions On Circuits And Systems II: Express Briefs*, vol. 54, no. 11, pp. 974–978, Nov. 2007.

[19] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhauser Boston, 2009.

H. Fatih Ugurdag received the BS degrees in EE and Physics from Boğaziçi University in Istanbul, Turkey, both in 1986, and the MS and PhD degrees in EE from Case Western Reserve University in Cleveland, Ohio in 1989 and 1995, respectively. He started his academic career in 2004. He has been with Ozyegin University, Istanbul, Turkey since 2010, where he is an associate professor. He has also been a consultant at Vestek R&D Corp. since 2007. He worked full-time in the industry in the US between 1989 and 2004. His employers included GM, Lucent, Juniper, and Nvidia. Prof. Ugurdags research interests are computer arithmetic, reconfigurable computing, design automation, real-time video processing, optical wireless communications, and automotive embedded systems.

Florent de Dinechin, born 1970, obtained a master's degree from cole Normale Supérieure de Lyon in 1993, then a PhD from Université of Rennes-1 in 1997. After a postdoctoral position at Imperial College, London, he joined École Normale Supérieure de Lyon as an assistant professor, then INSA-Lyon as a professor. His research interests include hardware and software computer arithmetic, FPGA arithmetic, FPGA computing, floating-point, formal proofs for arithmetic algorithms, elementary function evaluation, and digital signal processing. Since 2008, he manages the FloPoCo software project.

Y. Serhan Gener received the BS degree in Computer Engineering from Yeditepe University, Istanbul, Turkey in 2015. He is currently working towards the MS degree and is a graduate assistant in Department of Computer Engineering at Yeditepe University.

Sezer Gören received the BS and MS degrees in EE from Boğaziçi University, Istanbul, Turkey, and the PhD degree in computer engineering from the University of California, Santa Cruz. She was a senior engineer in Silicon Valley from 1998 to 2004 at Syntest, Cadence, Apple, PMCSierra, and Aarohi Communications. She is currently a professor in the Computer Engineering Department at Yeditepe University, Istanbul, Turkey. Her research interests include reconfigurable computing, design automation, design verification, test, computer arithmetic, and embedded system design.

Laurent-Stéphane Didier was born in Marseille, in 1970. He received his MSc from *Ecole Normale Supérieure de Lyon*, France in 1994, his PhD from the University of Marseille in 1998. He has been an associate professor at the *Université de Bretagne Occidentale*, Brest and *Université Pierre et Marie Curie*, Paris. His is now a professor at *Université de Toulon*, France. His research interests include computer arithmetic, efficient and safe implementation of cryptographic operators, computer architecture, and FPGAs.